UNIT 2. GREEDY ALGORITHMS

1. Introduction.

The greedy algorithms are one of the simplest and at the same time the most used schemes. It is typically used to solve optimization problems. The conditions that are met are:

- There is an entry of size n that are candidates to be part of the solution;
- There is a subset of those n candidates that meet certain restrictions: it is called feasible solution;
- It is necessary to obtain the feasible solution that maximizes or minimizes a certain objective function: it is called an optimal solution.

The greedy scheme proceeds in steps:

- Initially the set of chosen candidates is empty.
- In each step, we try to add to the set of the chosen ones "the best" of the non-chosen ones (without thinking about the future), using a selection function based on some optimization criterion (it may or may not be the objective function).

After each step, it is necessary to see if the selected set is complete (that is, if a solution can be reached adding more candidates);

- If the set is not complete, the last chosen candidate is rejected and it is not considered again in the future;
- If it is complete, it is incorporated into the group of chosen ones and always remains in it;

After each incorporation, it is checked whether the resulting set is a solution, so that the algorithm ends when a solution is obtained. It is considered that the algorithm is correct if the solution found is always optimal. However, although the greedy algorithms are usually efficient and easy to design and implement, not all the problems can be solved using greedy algorithms, since sometimes they do not find the optimal solution or even find no solution when the problem has one.

The scheme of the greedy algorithms is the following:

```
function greedy(C: set): set
    {C is the set of candidates}
    S:= ∅
    while C≠∅ and no solution(S)
            x:=element of x which maximizes select(C)
            C:=C-{x}
            if feasible{SU{x}} then S:= SU{x}
    if solution(S) then return S
    else return “no solutions”
```

2. The problem of changing money.

The objective of this problem is to return a quantity of money with the least possible number of coins, starting from a set of valid types of coins, of which there is supposed to be enough quantity to make the breakdown, and from an amount to be returned. The fundamental elements of the scheme are:

- Set of candidates: each of the currencies of the different types that can be used to make the breakdown of the given amount.
- Solution: a set of coins returned after the breakdown and whose total value is equal to the amount to be broken down.
- Completable: the sum of the values of the coins chosen at a given time does not exceed the amount to be broken down.
- Selection function: choose if possible the currency with the highest value among the candidates.
- Objective function: total number of coins used in the solution (must be minimized).

Basic scheme:

```
function return change(n): set of coins
    {Return the change of n units using the least number possible of coins. Constant
    C especifies the avalaible coins}
    const C=[100,25,10,5,1}
    S≠∅ {S is a set that will contain the solution}
    s=0 {s is the sum of the elements of S}
    while s≠n do
        x:=the highest element in C such that s+x≤n
        si that element does not exist then return "no solution found"
        S=SU{a coin of value x}}
        s=s+x
    return S
```

3. Problem of the backpack.

In this problem we have a backpack that supports a maximum weight W and there are n objects that we can load in the backpack, each object has a weight w_i and a value v_i. The objective is to fill the backpack maximizing the value of the objects it transports. We assume that objects can be broken, so that we can carry a fraction x_i ($0 \leq x_i \leq 1$) of an object.

Mathematically it is about maximizing $\Sigma x_i v_i$ (value of the load) with the constraint $\Sigma x_i w_i \leq W$ (weight of the load less than the total weight) where $v_i > 0$, $w_i > 0$ and $0 \leq x_i \leq 1$ for $1 \leq i \leq n$.

Basic scheme:

```
Function backpack (w[1..n],v[1..n],W):matrix[1..n]
    {Inicialization}
    for i=1 to n do x[i]:=0
    weight=0
    {greedy loop}
    while weight<N do
        i:=the best remaining element
        if weight+w[i]≤W then
            x[i]:=1
            weight:= weight +w[i]
        else
            x[i]:=(W- weight)/w[i]
            weight:=W
    return x
```

Example: n=5; W=100.

	1	2	3	4	5	
v_i	20	30	66	40	60	
w_i	10	20	30	40	50	$\sum_{i=1}^{n} w_i > W$

When selecting there are three possibilities:

1. Most valuable object? ↔ v_i max
2. Lighter object? ↔ w_i min
3. More profitable object? ↔ v_i / w_i max

Depending on the case, the value of the objective function will change:

	1	2	3	4	5	
v_i	20	30	66	40	60	
w_i	10	20	30	40	50	
v_i/w_i	2.0	1.5	2.2	1.0	1.2	Objetive $\sum_{i=1}^{n} x_i w_i$
xi (v_i max)	0	0	1	0.5	1	146
xi (w_i min)	1	1	1	1	0	156
xi (v_i/w_i max)	1	1	1	0	0.8	164

Therefore, the first strategy is to choose the object with the greatest total benefit. However, it could be that his weight is excessive and that the backpack is filled very quickly with small total benefit. In the second strategy the object that weighs less is chosen, to accumulate benefits of a greater number of objects. However, it is possible that objects with small benefit are chosen simply because they weigh less. The third strategy, which is the optimal one, is to take the objects that provide the greatest benefit per unit of weight. It is important to note that the algorithms resulting from applying either of the first two strategies are also greedy, but they do not calculate the optimal solution.

Therefore, the greatest difficulty of a greedy algorithm is to correctly choose the selection function.

If we calculate the efficiency of the algorithms, there are two possibilities:

1) Direct implementation: in this case the loop requires a maximum of n (number of obj.) Iterations: O(n) (one for each possible object in the case that all fit in the backpack) and the selection function requires searching the object with the best value / weight ratio: O(n). Therefore, the cost of the algorithm is

$$O(n)\cdot O(n)=O(n^2)$$

2) Implementation by pre-ordering the objects: first the objects are ordered from highest to lowest value / weight ratio (there are ordering algorithms O(nlogn)). With the ordered objects the selection function is O(1). Therefore, the cost of the algorithm is

$$O(\max(O(n\log n), O(n)\cdot O(1)))=O(n\log n)$$

4. Minimum coating trees

The objective of this problem is, starting from a given graph, to obtain a new graph that only contains the necessary edges for a global optimization of the connections between all the nodes, understanding as "global optimization" that some pairs of nodes may not be connected between them with the minimum possible cost in the original graph. Normally this problem applies to situations that have to do with geographical distributions, such as, for example, a set of computers geographically distributed in different cities of different countries to which you want to connect to exchange data, share resources, etc.; request to the different telephone companies the rental prices of lines between cities; or ensure that all computers can communicate with each other, minimizing the total price of the network.

Important definitions for this problem:

1) Free tree: is a non-directed, connected and acyclic graph
- every free tree with n vertices has n-1 edges
- if an edge is added, a cycle is introduced
- if an edge is deleted, vertices are not connected
- any pair of vertices is linked by a single simple path
- a free tree with a distinguished vertex is a tree with a root

2) Coating tree of a non-directed graph and not negatively labeled: it is any subgraph that contains all the vertices and that is a free tree.
3) Minimum cost coating tree: it is a coating tree and there is no other coating tree whose sum of edges is smaller.

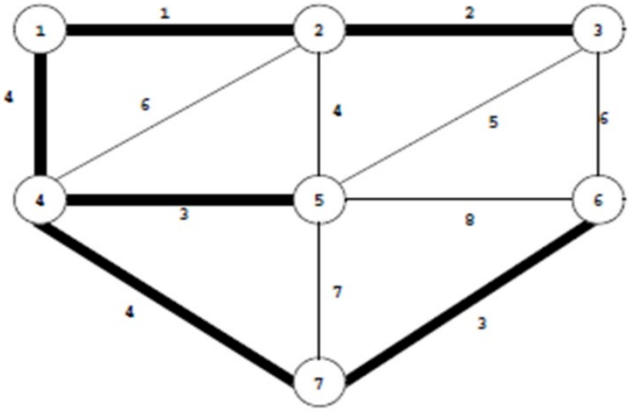
Fundamental property of minimum cost coating trees: "Let g be a non-directed graph connected and not negatively labeled, $g\in\{f: VxV \rightarrow E\}$, let U be a set of vertices, $U\subset V$, $U\neq\emptyset$, if $< u, v>$ is the smallest edge of g such that $u\in U$ and $v \in V \setminus U$, then there is some minimum cost covering tree of g that contains it ".

Starting from this property, a greedy algorithm can be designed to solve the problem, whose fundamental elements are:

- Objective function: to minimize the length of the covering tree.
- Candidates: the edges of the graph.
- Function solution: minimum length coating tree.
- Feasible function: set of edges that does not contain cycles.
- Selection function: varies with the algorithm;
  -Select the lower weight edge that has not yet been selected and that does not form a cycle (Kruskal Algorithm).
  - Select the lowest weight edge that has not yet been selected and that forms a tree along with the rest of the selected edges (Prim Algorithm).

4.1. Kruskal algorithm.

The main idea is selecting the edge of less weight that has not yet been selected and that does not connect two knots of the same connected component, that is, that does not form a cycle. Example:



The algorithm would operate as indicated in the following table:

Step	Considered edge	Connected components
-	-	{1},{2},{3},{4},{5},{6},{7}
1	{1,2}	{1,2},{3},{4},{5},{6},{7}
2	{2,3}	{1,2,3},{4},{5},{6},{7}
3	{4,5}	{1,2,3},{4,5},{6},{7}
4	{6,7}	{1,2,3},{4,5},{6,7}
5	{1,4}	{1,2,3,4,5},{6,7}
6	{2,5}	forma ciclo
7	{4,7}	{1,2,3,4,5,6,7}

In the form of a pseudocode:

```
function Kruskal (G=<N,A>: graph, length: A→R+): set of edges
    {Inicialization}
    Order A by growing lengths
    n:= number of nodes in N
    T≠∅ {will contain the edges of the mínimum coating tree}
    Inicialize n sets, each one containing a different element from N
    {greedy loop}
    repeat
        e:= (u,v) {shortest edge, not yet considered}
        compu:=search(u)
        compv:= search (v)
        if compu≠compv then
            fuse(compu,compv)
            T:=TU{e}
    until T has n-1 edges
    return T
```

Efficiency analysis: we assume that |N|=n^|A|=a. Then the cost of each operation is:

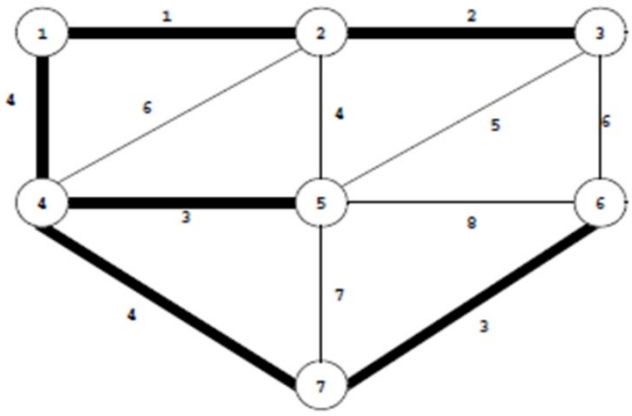
- Sort A: O(aloga)≡O(alogn) since n-1≤a≤n(n-1)/2
- Initialize n disjoint sets: O(n)
- For search and merge operations: O(alogn)
- For the other operations: O(a).

So:

$$T(n)=O(alogn)$$

4.2. Prim algorithm.

In this case, it is a matter of starting with any node, selecting the edge with the least weight that has not yet been selected and that forms a tree along with the rest of the selected edges.



Prim algorithm repeatedly applies the property of minimum cost for coating trees incorporating an edge at each step. A set B of treated vertices is used and the minimum edge joining a vertex of B with another of its complement is selected in each step.

- Initialization: B = {arbitrary node of N} = {1},
  T empty.
- Selection: shorter edge than part of B:
  (u, v), u∈B ^ v∈N \ B → is added (u, v) to T and v to B
- T defines at all times a minimum expanded tree of the subgraph (B, A)
- End of the algorithm: B = N. At this point, T offers an optimal solution to the problem.

Therefore, the Kruskal algorithm is a forest that grows, instead, Prim algorithm is a single tree that grows until reaching all the nodes and calculates the minimum expanded tree. For example:

Step	Selection	B
ini	-	{1}
1	{1,2}	{1,2}
2	{2,3}	{1,2,3}
3	{1,4}	{1,2,3,4}
4	{4,5}	{1,2,3,4,5}
5	{4,7}	{1,2,3,4,5,7}
6	{7,6}	{1,2,3,4,5,6,7}

The implementation of Prim algorithm would be:

```
function Prim (L[1..n, 1..n]): set of edges
    {Inicialization: just node 1 is in B}
    T≠∅ {will contain the edges of the mínimum coating tree}
    for i:=2 to n do
        closest[i]:=1
        distmin[i]:=L[i,1]
    {greedy loop}
    repeat n-1 times
        min:=∞
        for j:=2 to n do
            if 0≤distmin[j]<min then
                min:=distmin[j]
                k:=j
        T:=TU{closest[k],k}
        distmin[k]
        for j:=2 to n do
            if L[j,k]<distmin[j] then
                distmin[k]:=L[j,k]
                más próximo[j]:=k
    return T
```

Calculating its efficiency, we have two phases:

- Initialization = O(n)
- Greedy loop: n-1 iterations, each for nested O(n). Total: $T(n)=O(n^2)$

Therefore, with more efficient data structures (mound) could be improved to O(alogn), like Kruskal, as shown in this table:

	Prim $O(n^2)$	Kruskal O(alogn)
Dense graph: a→n(n-1)/2	**$O(n^2)$**	$O(n^2 \log n)$
Disperse graph: a→n	$O(n^2)$	**O(nlogn)**

5. Minimum paths in graphs.

In this problem, we start with a graph labeled with non-negative weights and try to find the minimum length path between two given vertices, understanding as length or cost of a path sum of the weights of the edges that compose it.

5.1. Dijkstra algorithm.

The problem consists in given a graph G=(N,A) directed, with lengths in the edges ≥ 0, with a node distinguished as the origin of the roads (node 1), finding the minimum paths between the origin node and the rest of the nodes of N. For this, it is assumed that the minimum path of a node to itself has zero cost and a value ∞ at the position w of the vector indicates that there is no path from v to w.

The algorithm is used for directed graphs (although the extension to non-directed is immediate). The algorithm generates one by one the paths of a node v to the rest in order of increasing length and uses a set of vertices where, at each step, the nodes are saved for those whose minimum path is already known. In each step, the algorithm returns a vector indexed by vertices: in each position w the cost of the minimum path connecting v and w is saved, so that each time a node is incorporated into the solution, it is checked whether the paths are not yet definitive can be shortened by going through it.

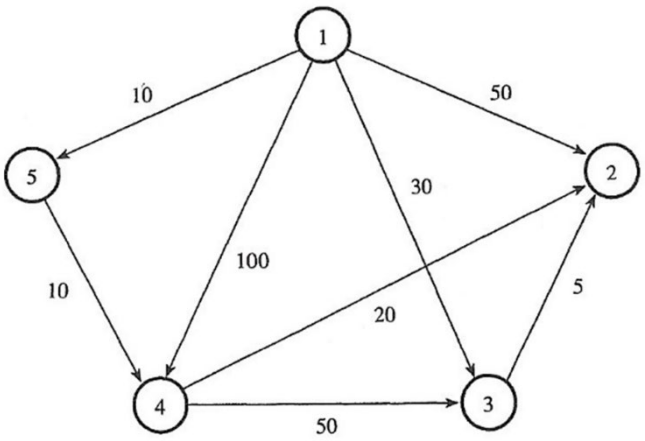
The greedy technique consists of the following:

- 2 sets of nodes: S: selected (minimum path established); C: candidates (others).
- Invariant: N = S∪C
- Initially S = 1 → final: S = N: solution function.
- Selection function: C node with the lowest known distance from 1 → there is provisional information about minimum distances.

The implementation would be:

```
function Dijkstra (L[1..n, 1..n]): matrix[2..n]
    matrix D[2..n]
    {Inicialization}
    C:={2,3,..,n}{S=N\C only exists implicitly}
    for i:=2 to n do D[i]:=L[1,i]
    {greedy loop}
    repeat n-2 times
        v:=an element from C which minimizes D[v]
        C:=C\{v} {and implícitly S:=S\{v}}
        for each w∈C do
            D[w]:=min(D[w],D[v]+L[v,w])
    return D
```

Example:



Step	Selection	C	D[2]	D[3]	D[4]	D[5]
Ini	-	{2,3,4,5}	50	30	100	**10**
1	5	{2,3,4}	50	30	**20**	10
2	4	{2,3}	40	**30**	20	10
3	3	{2,}	35	30	20	10

6. Planification problems.

In this section there are two types of problems related to planning tasks in a single machine. On the one hand, it is about minimizing the average time that each task invests in the system and, on the other hand, the tasks have a fixed execution period, and each task brings certain benefits only if it is finished when its term arrives. The objective in both cases is to maximize profitability.

6.1. Minimization of the time in the system.

Suppose we have three clients with t_1=5, t_2=10 and t_3=3. There are six possible service orders.

Order	T	
123:	5+(5+10)+(5+10+3)=38	
132:	5+(5+3)+(5+3+10)=31	
213:	10+(10+5)+(10+5+3)=43	
231:	10+(10+3)+(10+3+5)=41	
312:	3+(3+5)+(3+5+10)=29	→ optimum
321:	3+(3+10)+(3+10+5)=34	

Obviously, in this case, optimal planning is obtained when the three customers are served in order of increasing service times: customer 3, who needs the shortest time, is served first, while customer 2 who needs more time is served, is the last one to be served. If we serve customers in descending order of service times, the worst planning is obtained. Therefore, a greedy algorithm that constructs optimal planning element by element, all that needs is to order the clients in order of non-decreasing time of service, which requires a time that is in O(nlogn).

6.2. Planning with fixed term.

In this case the system has to execute a set of n tasks, each of which requires a unit time. At any moment we can execute only one task. Task i produces benefits $g_i > 0$ only in the case that it is executed at a time before d_i.

For example, with n=4 and the following values:

i	1	2	2	4
g_i	50	10	15	30
d_i	2	1	2	1

The plans that must be considered and the corresponding benefits are:

Secuence	Benefit	Secuence	Benefit	
1	50	2,1	60	
2	10	2,3	25	
3	15	3,1	65	
4	30	4,1	80	→ optimum
1,3	65	4,3	45	

The sequence 3,2 is not considered because task 2 would be executed at time t=2, after its term which is d_2 = 1. To maximize our benefits, we should execute the planning 4,1.

It is said that a set of tasks is feasible if there is at least one sequence that allows all the tasks in the set to be executed before their respective deadlines. A greedy algorithm consists of constructing the planning step by step, adding in each step the task that has

the highest value of g_i among those that have not yet been considered, as long as the set of selected tasks remains feasible.

In the example, we first select task 1. Then task 4; the set {1,4} is feasible because it can be executed in the order 4,1. The set {1,3,4} is not feasible, therefore the task 3 is rejected. The set {1,2,4} is also not feasible and task 2 is rejected. Our solution, in this case optimal, is to execute the set of tasks {1,4}, which can only be done in the order 4,1.

Let J be a set of k tasks. Suppose that the tasks are numbered in such a way that $d_1 \leq d_2 \leq ... \leq d_k$. Then the set J is feasible if and only if the sequence 1,2, ... k is feasible. In this case, the greedy algorithm always finds an optimal planning.

To implement the algorithm, suppose that the tasks are numbered in such a way that $g_1 g_2 ... g_n$. We assume that $n > 0$ and $d_i > 0$ for $1 \leq i \leq n$, and that an additional space is available at the beginning of the vectors d (containing the terms) and j (where the solution is built). These additional cells are called "sentinels". By storing an adequate value in the sentinels, repetitive checks of ranges that consume a lot of time are avoided. The pseudocode of the algorithm is as follows:

```
function secuence (d[0..n]): k,matrix[1..k]
    matrix j[0..n]
    {The planning is built step by step in matrix j. The variable k says how many tasks
    are already in the planning}
    d[0]:=j[0]:=0 {sentinels}
    k:=j[1]:=1 {task 1 is always selected}
    {greedy loop}
    for i:=2 to n do {decreasing order of g}
        r:=k
        while d[j[r]]>max(d[i],r) do r:=r-1
        if d[i]>r then
            for m:=k step -1 to r+1 do j[m+1]:=j[m]
            j[r+1]:=i
            k:=k+1
    return k,j[1..k]
```

The tasks of matrix j are in increasing order of time. When task i is being considered, the algorithm checks whether it can be inserted into j in the right place without carrying out any task that is already in j beyond its term. If so, i is accepted, otherwise rejected. Example with six tasks:

i	1	2	2	4	5	6
g_i	20	15	10	7	5	3
d_i	3	1	1	3	1	3

Efficiency of the algorithm: the ordering of tasks in decreasing order of benefit requires a time that is in O(nlogn). The worst case is when you classify the tasks in descending order of terms, and when all of them have a place in planning. In this case, when the task is being considered, the algorithm examines the k=i-1 tasks that are already planned, to find a place for the newcomer, and then moves a whole place. There are $\sum_{k=1}^{n=1} k$ passed

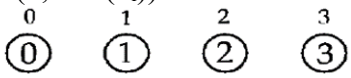
through the loop **while** and $\sum_{m=1}^{n=1} m$ passed through the loop **for** internal. Therefore, the complexity is $O(n^2)$.

A faster version of the algorithm: it is assumed that the label of the set produced by a merge operation is necessarily the label of one of the sets that have been merged. The planning produced in the first place may contain gaps, the algorithm ends up moving tasks forward to fill them:

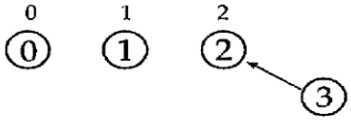
```
function secuence2 (d[0..n]): k,matrix[1..k]
    matrix j,F[0..n]
    p=min(n,max{d[i]|1≤i≤n})
    for i:=0 to p do
        j[i]:=0
        F[i]:=1
        inicialize set {i}
    {greedy loop}
    for i:=1 to n do {decreasing order of g}
        k:=search(min(p,d[i]))
        m:=F[k]
        if m≠0 then
            j[m]:=i
            l:=search(m-1)
            F[k]:=F[l]
            fuse(k,l) {the resulting set has the label k or l}
    k:=0
    for i:=1 to p do
        if j[i]>0 then
            k:=k+1
            j[k]:=j[i]
    return k,j[1..k]
```

If we are given the problem with tasks already ordered by decreasing benefits, such that it is possible to obtain an optimal sequence by calling the previous algorithm, then most of the set will be invested in manipulating disjoint sets. It is necessary to perform a maximum of 2n search operations and n merge operations, then the time required is in $O(n\alpha(2n,n))$ where α is the slow growth function. If the tasks are in an arbitrary order, we must first order them, and obtaining the initial sequence requires a time O(nlogn). Returning to the example of the six tasks:

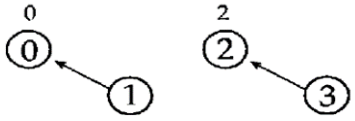
Start: p=min(6,max(d_i))=3



Trial 1: d_1=3, task 1 is assigned to position 3

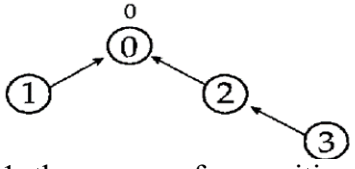


Trial 2: d_2=1, task 2 is assigned to position 1



Trial 3: d_3=1, there are no free positions because F=0

Trial 4: d_4=3, task 4 is assigned to position 2



Trial 5: d_5=1, there are no free positions

Trial 6: d_6=3, there are no free positions